The LIGHTest Foundation

Mödersheim, Sebastian Alexander; Schlichtkrull, Anders

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

The LIGHT^{est} Foundation

DTU Technical Report-2018-6

Sebastian Mödersheim and Anders Schlichtkrull *

November 20, 2018

Abstract

This technical report presents the Trust Policy Language (TPL), its semantics, and shows how it can be used to specify policies for trust, trust translation, and delegation. TPL and its semantics are highly inspired by Prolog. The semantics come in two flavors, an executable semantics and a logical semantics. Because of the executable semantics, a specification in TPL can also be executed as a program that checks if a decision lives up to a policy. TPL is the language used in LIGHT^{est}, which is a toolbox for digital policies for trust, trust translation, and delegation. TPL serves as its foundation and its semantics defines the meaning of a specified policy is. The semantics can therefore be used to settle any discussion of what the meaning of a specified policy is. In order to make TPL easy to use, we present GTPL which is a graphical language that allows end users to specify policies in TPL.

1 Introduction

This technical report represents the formal basis of the LIGHT^{est} project[1]. It does so by describing the LIGHT^{est} Trust Policy Language (TPL) language and how it can be used to define and describe policies for trust, trust translation, and delegation. While the syntax of a language defines which expressions it contains, the semantics of a language defines what they mean.

We strive for giving TPL a well-defined semantics where it is always clear exactly what the meaning of an expression is. This is important for two reasons. Firstly, we want to avoid that in some corner cases it is unclear what a particular policy means. With a clear semantics any disagreement on the meaning of a policy can be resolved by referring to the semantics. This is particularly important in the LIGHT^{est} project which brings together partners with different backgrounds from both academia and industry. Discovering and resolving such mismatches early in the project is invaluable. Secondly, we want to mechanize policies, i.e., have computer programs called automated trust verifiers (ATVs) determine whether a given decision satisfies a given policy or not. We therefore need a way to turn trust policies into algorithms that the ATVs can follow. With a precise semantics of the policy language it is possible to prove that an algorithm correctly implements a policy, or even better, automatically derive the algorithm from the policy. In fact, inspired from this latter point we have decided to let TPL be a logic programming language based on the Prolog programming language, since this allows for declarative policy specifications that are directly executable. In other words, TPL can be seen as both a specification language and as a programming language.

*We thank Rasmus Birkedal for contributing to an early version of this work.

[1] See `https://www.lightest.eu` and `https://www.lightest-community.org`.

TPL should not be usable by only programmers, and we therefore also introduce a simple graphical language for end users that can handle most of the use cases of LIGHT^{est} that we foresee and that can be easily mapped into TPL.

2 TPL Design Decisions

The LIGHT^{est} Trust Policy Language, or TPL for short, is a language for defining policies for trust, trust translation, and delegation. TPL allows us to integrate these different parts of the LIGHT^{est} project in a precise and uniform way and its semantics can serve as an unambiguous reference for testing and formally proving the correctness of any implementations based on TPL such as an ATV.

TPL is a subset of Prolog that abandons some of the more advanced features in favor of achieving a simpler semantics. Like in Prolog, a program is a list of definite Horn clauses and TPL shares most of the syntax of Prolog. This is inspired by similar languages for access control based on Horn clauses such as SecPAL and DKAL [1, 3].

The particular benefits from using Horn clauses are the following:

- Lists of Horn clauses have a simple *logical* semantics, namely the semantics of first-order logic.
- Horn clauses famously have an *executable* semantics. With this semantics a list of Horn clauses is a program. This means that a TPL policy immediately gives rise to an algorithm for evaluating whether a decision lives up to the policy.
- More generally, the semantics of TPL hollows us to make queries of policies that go beyond evaluating whether a specific decision lives up to a given policy. We could, e.g. for a translation policy between levels of assurance from respectively an ISO standard and eIDAS, query which levels of assurance in the ISO standard correspond to eIDAS level "low". This technique can e.g. be used to discover ambiguities in policies.
- Prolog interpreters implement the *executable* semantics of TPL. We can use them as reference implementations of ATVs, for testing, or as the basis of implementations of ATVs.
- We can draw on previous experiences with and research on logic programming.
- Many policies are in practice just simple enumerations of cases. These are trivial to express as Horn clauses.
- On the other extreme, with the executable semantics, Horn clauses are Turing complete. Therefore every algorithm can be written using Horn clauses and executed following the executable semantics. This means that the language will never limit us by lack of expressive power.
- Lists of Horn clauses are suitable for expressing many concepts that regularly arise in policies for trust. We give two examples: Firstly, one can easily describe the logical relationships between several criteria, for instance, that when an order is above a certain value then stricter criteria have to be fulfilled. Secondly, Horn clauses are suitable for describing delegation, for instance where documents are signed by the proxy of a company.
- It is possible to extend the *executable* semantics with a number of predefined predicates that trigger the necessary queries to servers, e.g. using DNSSec, and process their answer. When we make such extensions, we must consider what this means for the logical semantics. Here we can again rely on previous experiences with and research on logic programming.

- For the average users we can provide design patterns for their policy.
- For the average users we can provide an interface to a graphical language which is more intuitive to use but has limited expressive power.

TPL allows for complex forms of reasoning about trust, for instance, trust translation and delegation. It is, however in the hands of each business to set the policy what they are willing to accept and we expect that in the vast majority of cases simple policies will be sufficient. For both complex and simple policies, it is important that we provide a specification language that is both clear and unambiguous, and where the policies can be evaluated by an ATV. In this technical report we argue that TPL has these properties.

3 A Gentle Introduction to TPL

We illustrate the flavor of TPL and what specifications could look like with a few examples. The language is based on definite Horn clauses, i.e. formulae of the form

conclusion :− requirement1, ..., requirementN.

One may simply read `:-` as the word "if" and the commas as "and". The logical meaning of a definite horn clause is that the conclusion holds, if all the requirements hold. We will also call the conclusion the *left-hand side* or *head* of the clause, and the requirements the *right-hand side* or *body* of the clause. Note that this is a logical implication, not a biimplication: if the requirements do not hold, the clause does not tell us whether the conclusion holds or not.

Consider as a specific example the Horn clause

trust(X) :− delegate_of(X, Y), trust(Y).

This can be interpreted as "I trust X if X is a delegate of an entity Y that I trust." Here, we have introduced as new vocabulary the *predicates* trust and delegate_of that have no built-in meaning in TPL; in fact, they obtain their meaning from the clauses we specify. X and Y are *variables* which are a subclass of the more general notion of *terms*. A term is either a variable (such as X), a constant (such as a, b or c) or a function applied to other terms (such as $f(X, Y)$ or $g(g(X), f(x))$). A variable such as X and Y can be replaced by any term and the clause applies. For instance, suppose that trust(a) and delegate_of(b, a) already hold, then we can derive trust(b), because we can *instantiate* X to b and Y to a and apply the rule.

In fact, we will typically have a basis of clauses like trust(a) that already hold initially; they are written as *facts*, i.e., they have no right-hand side:

trust(a).
delegate_of(b, a).

Suppose we also add delegate_of(c, b) and delegate_of(c, d), then we can also derive trust(c) and trust(d), i.e., we can form arbitrary long chains of delegation. If this is not desired in a policy, one can introduce different predicates for trust, distinguishing whether someone is trusted directly or through delegation. For the sake of this example, let us use trust(X) for the direct trust and introduce a new predicate trustD(X, N) for trust through delegation; here N denotes the length of the chain of delegation, e.g., trust(X) is equivalent to trustD(X, 0). The length of a chain of delegations is called a delegation level. Then we can specify trust by delegation as the following Horn clauses:

trustD(X, 0) :− trust(X).
trustD(X, N) :− N > 0, delegate_of(X, Y), trustD(Y, N − 1).

Thus we can now derive in this example:

```
trustD(a, 0).
trustD(b, 1).
trustD(c, 2).
trustD(d, 2).
```

Be aware that in contrast to many Prolog interpreters, the current version of TPL considers N − 1 to be the result of subtracting 1 from N. We elaborate on this in subsection 4.2.

We can now easily express a policy for accepting an electronic purchase based on both the delegation level and the amount of a purchase, specifically the policy that below 100 Euro any delegation level is fine, below 1000 Euro the delegation level should be at most 1, and up to 1 Mio. Euro we do not accept delegation. This is specified as follows:

```
order(X, M) :- M =< 100, trustD(X, N).
order(X, M) :- M =< 1000, trustD(X, N), N =< 1.
order(X, M) :- M =< 1000000, trustD(X, 0).
```

We use the less-than-or-equals predicate (=<). Note that logically the order of the clauses does not matter, and neither does the order of the requirements in the body of a clause. However, in a typical Prolog interpreter this order does matter and we shall use this now as a glimpse into the typical working of an interpreter.

The interpreter is fed with a rule base, a list of definite Horn clauses and a *query* which can be any predicate (even containing variables). For instance consider the query order(a, 300). The interpreter goes through the clauses and takes the first one whose left-hand side matches the query; in our case that is the clause for purchases up to 100 Euro. It now checks the requirements in the order they are specified. In the example this fails at the first requirement since the query is for more than 100 Euro. In such a case the interpreter continues with the next matching clause, here the one for 1000 Euro; the first requirement is obviously satisfied. For the second one, the interpreter must start a new query, namely whether trustD(a, N). Note that this query now contains a variable, and thus we are asking if a is trusted in *any* number of delegations. Indeed that will return trustD(a, 0) and thus set N = 0. With this, also the third requirement of the clause is met.

We are also able to formulate queries like trustD(X, 2) which corresponds to the question "Who do we trust on delegation level 2?" We obtain two answers, namely X = c and X = d. This way we can use the interpreter also to reason about policies and make policy testing, e.g., we can test that the policy indeed reflects what we wanted it to express.

4 A Detailed Definition of TPL

TPL consists of three parts:

- The core language.
- The language extensions.
- The standard library.

In this section we will describe these three parts.

4.1 The core language of TPL

The core language of TPL is similar to Prolog but has a number of restrictions. The first one is on the syntax:

Prolog is quite liberal on the use of function and predicate symbols—there is no type system, there is no distinction between function and predicate symbols, and symbols can be used with different arity (number of arguments) throughout the code. We deliberately restrict this here as it can lead to bugs that are hard to find, but does not usually make specification easier, thus we define that we have the following disjoint sets of symbols:

- Variable symbols: they start with upper-case letters.
- Function symbols: they start with lower-case letters and have a fixed arity. Constants are a special case that have zero arguments.
- Predicate symbols: they also start with lower-case letters and have a fixed arity.

Terms are now inductively defined as the least set that contains variables, and for every function symbol f of arity n, if $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. Further if $t_1, \ldots, t_n$ are terms and p is a predicate symbol of arity n, then $p(t_1, \ldots, t_n)$ is a predicate. A clause is of the form p:– $p_1, \ldots, p_n$. where p and the p_i are predicates. Here we will usually require that the variables occurring in p are a subset of the variables occurring in the p_i.

We do not require the user to declare the function and predicate symbols explicitly; instead this is inferred by the interpreter and checked for consistency in the specification. For example, if the specification contains the fact signature(privkey(alice), X) then the interpreter infers that signature is a binary predicate symbol, privkey is a unary function symbol, alice is a constant and X is a variable, and the interpreter checks that this is consistent—i.e. that signature is always used as a binary predicate symbol, privkey always as a unary function symbol, alice always as a constant.

We here briefly sketch two ways to formally define the semantics of TPL—the first one is executable and the second one is logical. We only sketch them since the semantics of Prolog has been described in detail.

Executable Semantics The executable semantics of TPL is the same as that of Prolog except that in TPL our unification always includes the so-called occurs check. The semantics of Prolog can be described as an interpreter. This has been done elsewhere and we refrain from repeating the semantics here and instead refer to the description by Deransart, Ed-Dbali and Cervoni [2][Section 4.2].

Logical Semantics A logical view of the semantics can be obtained if we consider the Horn clauses as logical formulas of first-order logic, where :– is $\leftarrow$ (logical implication from right to left), the comma is logical conjunction and all variables of every Horn clause are universally quantified, e.g., p(X, Y) :– q(X) becomes $\forall X, Y.\ q(X) \rightarrow p(X, Y)$.

Then, given a set of Horn clauses H and a query $q_1, \ldots, q_n$, the solutions are those substitutions σ of the variables in the q_i such that $H \models \sigma(q_1), \ldots, \sigma(q_n)$ where $\models$ represents the semantics of first-order logic as defined in any standard text book. The advantage of this logical formulation is that it is independent of the procedural aspects like the order of the clauses or termination issues.

4.2 The Language Extensions of TPL

We have considered a number of features—inspired by Prolog—that extend the core language of TPL. The included extensions are

- List syntax.
- Arithmetic.

List Syntax TPL has, like Prolog, special syntax for lists:

- [] denotes the empty list.
- [X|Xs] denotes the list which has X as the first element and the elements in Xs as the rest of the list in the same order as in Xs.

This is a very conservative extension since without it, [] could simply be represented by a ground term empty and [X|Xs] using a function application as cons(X, Xs).

Arithmetic TPL contains built-in datatypes for arithmetic. The symbols used are the following:

Symbol	Meaning
$+$	Addition
$-$	Subtraction
$*$	Multiplication
$/$	Division
$<$	Less than
$>$	Greater than
$=<$	Less than or equals
$>=$	Greater than or equals

The operators are defined for integers and floating point numbers. Prolog interpreters typically impose restrictions requiring that some of the operators are only ever applied to ground (variable free) terms or terms that are fully instantiated to ground (variable free) terms in the current execution. TPL imposes similar restrictions.

Prolog interpreters typically consider $+$, $-$, $*$, and $/$ to be function symbols on line with any other function symbols; the idea is that e.g. (N+1)−(M∗2) is a term in the same way as s(a(N,1),m(M,2))—the former just happens to represent an arithmetic expression. A term containing these symbols can then be evaluated using a function called is. In the current version of TPL, we deviate from this approach and let the application of these operators directly represent the result of the expression. Implementing this in an interpreter is non-trivial when variables occur that have not yet been fully instantiated to ground terms, however, when no such variables occur in the current execution it could be done by evaluating the arithmetic expression; this could e.g. happen before unification. There are possibilites for broadening this approach to more general fragments such that all variables in the context do not need to be fully instantiated. As we improve TPL we will consider the consequences of the different ways of implementing arithmetics.

4.3 Standard Library

TPL comes with a standard library of predicates that can interact with servers. We will refrain from going into more detail here as it is still under development and we consider it out of scope for the present technical report.

5 Example: Entity Authentication Assurance Framework

Let us now give an example from the enrollment part of the ISO standard "ISO/IEC FDIS 29115: Entity Authentication Assurance Framework". The enrollment is specified by giving Horn clauses specifying the predicates loa1, loa2, loa3 and loa4, which represents levels of assurance 1 to 4 as well as Horn clauses specifying the predicates l2req, l3req and l4req, which represents the requirements introduced on levels 2 to 4.

Parts of this example require actions that, at least for now, are rarely expected to be performed by a computer, such as checking the physical passport of the person who shows up in person for enrollment. We express these actions as predicates without specifying them further.

We see two reasons for specifying an enrollment process in this way. The first one is formalization: by specifying the enrollment in a language with a formal semantics we can avoid the ambiguities that can occur in a natural language specification. The second one is computer automation: the enrollment process could, to some extent, be performed by a computer, since the unspecified predicates could later be defined to interface with databases containing the relevant information—e.g. a database containing information of who have shown up in person for enrollment.

We first specify three levels of assurance:

```
loa1(X) :- uniq(X).
loa2(X) :- loa1(X), l2req(X).
loa3(X) :- loa2(X), l3req(X).
loa4(X) :- loa3(X), l4req(X).
```

This expresses that an enrollment application satisfies level of assurance 1 if some predicate uniq is satisfied which represents that the identity of the person to be enrolled is unique. Any level above 1 is defined as living up to the requirements of the all lower levels and some additional requirements. For these we have the predicates l2req, l3req, and l4req which we specify in more detail:

```
l2req(X) :-
  person(X),
  inPerson(X),
  idDocument(X, D).
```

Here, person, inPerson and idDocument are predicates that refer to the condition that the entity to be enrolled is a person, showed up in person, and has a document D to prove their identity. There are two other ways to satisfy the level 2 requirements, namely, in the case of not showing up in person and in the case of not being a person at all:

```
l2req(X) :-
  person(X),
  notInPerson(X),
  idDocument(X,D).

l2req(X) :-
```

```
  nonPerson(X),
  autoritativeInformationRecorded(X).
```

In the level 3 requirements, we refer to the ID-document that the applicant has shown because the requirement is to check this document with the records of the original issuer who once produced this document.

```
l3req(X) :-
  person(X),
  inPerson(X),
  contactInformationVerified(X),
  idDocument(X, D),
  checkedWithSource(D),
  personalInformationCorroborated(X),
  verifiedCredentialClaim(X).
```

A similar modeling method we use here is to extract aspects of the application, e.g., that when the applicant does not show up in person, then they must have that they possess a level-3 certificate:

```
l3req(X) :-
  person(X),
  notInPerson(X),
  hasCredential(X, C),
  loa(C, L),
  L >= 3,
  verifiedCredentialClaim(X).
```

Here, to extract that certificate from the application we have introduced the predicate hasCredential(X, C) and from the certificate we extract the level of assurance using another predicate loa(C, L). This allows us to formulate the rules without specifying the precise structure of the terms X (the entire application) and C (the concrete credential).

Quite similarly we can now formulate that non-person entities need to apply with a level 3 credential that was issued by a human:

```
l3req(X) :-
  nonPerson(X),
  trustedHardwareUsage(X),
  hasCredential(X, C),
  loa(C, L),
  L >= 3,
  issuer(X, I),
  person(I).
```

An advantage of modeling the evaluation process abstractly with only loa1, loa2, loa3, loa4, l2req, l3req and l4req being specified by horn clauses is that the specification does not depend on a particular format of applications or credentials. For example, l3req is compatible with any credential technology for which one can specify a predicate loa(C, L) on credentials that extracts the level of assurance and produces failure when the credential in question does supply an assurance level.

6 Formats

The policies will be working on data with a variety of concrete data formats, from XML certificates, DNS resource records to custom data formats for electronic formats. In order to work with these data but avoiding low-level details like parsing into the policies, we use an abstract syntax interface to the concrete formats. This also allows one to use general solutions to prevent injection and overflow attacks entirely.

This interface between concrete syntax (the actual bytestrings) and the abstract syntax shall be a separate part of a TPL specification. This section outlines a module for TPL for specifying abstract and concrete syntax and their relationship. Basically, what one ultimately needs to implement is two things:

- a parser that reads a bytestring in concrete syntax and extracts the abstract syntax, and
- a pretty printer that produces from the abstract syntax the concrete syntax again.

First, for many formats (like XML) there are already implementations like XML libraries; given that they are implemented without any security problems, they can often serve as a good basis. What is often lacking, though, is a proper abstract data type for the parser to return. Also, one may implement a parser manually, but it is much more convenient to have a general method of specifying a description of concrete and abstract syntax that allows for an automated generation of the parser and pretty-printer.

To this end, we use an idea from the field of specifying security protocols. The idea is to combine the advantages of concrete and abstract syntax: from concrete syntax the advantage that we are completely precise on the used message formats and from abstract syntax the advantage that we avoid cluttering up the entire protocol description with complicated but largely irrelevant details about concrete syntax. As a real-world example, let us consider the first message of the TLS Handshake protocol. In a high-level description we would like to simply write a term like this:

$$\mathit{TLSclientHello}(T, R, S, \mathit{Cipher}, \mathit{Comp})$$

The actual message on the string level would be however:

```
'20' '3' '3' [ ['1' '3' '3' T R [S]1 [Cipher]2 [Comp]1 ]3 ]2
```

where `'n'` means a byte of value n and `[m]k` means that m is a message of a variable length, and this length is given as a k-byte field before m. (In the example, T and R have a fixed size.)

The idea is now that abstract function symbols like *TLSclientHello* are a sound abstraction of their concrete byte-level format if all formats we use are only fulfilling some reasonable properties [5]. With sound abstraction we mean: If the intruder can attack a system on the low byte level, then there is a similar attack on the abstract function level. Thus, if the abstract system is secure, then also the concrete system is. This means it is safe for us to just think in terms of the abstract level (in particular in modeling and verification).

The reasonable properties that this soundness result requires are:

- Each format f is *unambiguous*: if $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$ then $t_1 = s_1, \ldots, t_n = s_n$, i.e. there is no byte string that can be read in more than one way as format f.
- The formats are *pairwise disjoint*: if $f_1(t_1, ..., t_n) = f_2(s_1, \ldots, s_m)$ then $f_1 = f_2$ (and thus $n = m$, as well as $t_1 = s_1, \ldots, t_n = s_n$ by unambiguity), i.e., no byte string can be parsed as more than one format.

Let us consider one more example: many protocols exchange information using some XML-based formats. XML is itself a construction that allows arbitrary hierarchical structures and ensures unambiguity and disjointness, and will return the parsing result in abstract syntax, i.e., in that tree we have no longer any concrete syntax characters like angle brackets and slashes of an expression like <element>...</element>, but we rather see only a tree node of entity element and what its children are. Still, it is often nice to put yet another abstraction layer on top of such an XML format, so one does not have to browse such an XML parse tree but has a more immediate representation of the data that is suitable for one's purposes.

By using a well-written XML-library for parsing and pretty printing XML-formats, one can avoid many of the usual implementation problems, such as buffer overflows, "by construction". More generally, one of the ideas behind formats is that one should have a library of parsers/pretty printers, one for each format, and then use them, similar to a library of crypto functions. The point is in both cases that not every programmer needs to repeat all the common mistakes, but just reuse the best solution for a subtle programming problem.

There is some preliminary work on automatically generating said libraries of formats from description of formats supporting:

- XML-based formats.
- The ASN-style format description seen in the TLS example above.

Both of these are prototype implementations that support formats with a fixed number of elements [4].

6.1 Interfacing to TPL

Generally, we like to assume that a format consists of a number of attribute-value pairs and we can imagine this like a paper form that has several *fields*, where each field is clearly identified by an attribute name and one can fill in an attribute value. For instance an XML-based certificate may have the following format:

```
<cert>
  <firstname>Jane</firstname>
  <lastname>Doe</lastname>
  <dateofbirth>...</dateofbirth>
  ...
</cert>
```

Another format may structure the same information in a different way, e.g., like X.509 certificates where no explicit text identifies the fields like firstname, but the format itself defines which bytes of the text mean which field.

To work with such certificates in TPL without bothering about the concrete syntax of a format we assume to have a predicate extract (that is linked to a parser implementation) and that has three arguments: an instance of the format, an attribute, and the value for it, e.g., if C is a certificate of the sketched XML type, we could work with it in TPL as follows:

```
over18policy(C) :-
  extract(C, dateofbirth, D),
  today(T),
  addyear(D, 18, D2),
  D2 =< T.
```

where we assume today(T) is true for the current day at the execution of the policy, and addyear and =< work on arithmetic for the date datatype as expected.

Additionally, we shall assume that extract has a special attribute format that stores which kind of format a particular text is. For instance in the above we may additionally require on the right-hand side extract(C, format, myXMLcertificateType) where myXMLcertificateType is the identifier for our example XML certificate format. This special *format* field of course assumes that all formats are disjoint, i.e., that there is no bytestring that matches more than one format. In practice this might not always be the case, when in specific contexts some formats are used that are not disjoint to all formats of other contexts.

We have here assumed so far all formats to be essentially a list of attribute-value pairs. There are some extensions relevant in general: there may be optional attributes, and attributes where the value is itself a structured datatype, e.g., a list of arbitrary length. Note that all this can be implemented by corresponding parsers and pretty printers and the access through the extract predicate can be uniform.

An interesting extension though for TPL that arises from this is a type-system where we check that data is always handled with appropriate predicates. Such a type system is subject of future work.

7 Interfacing to DNSSec

The trust policies will often involve the requirement that the issuer of a given certificate is member of a particular trust list. This actually means that as part of checking the policy, one has to interact with a foreign server to obtain trust list entries. TPL allows us to formulate such a check as part of a policy, however such a mix of policy rules and interaction with third parties may raise some concern, and we will discuss this below in section **??**.

The general setup, as far as we are concerned here, is that trust lists are stored relative to a domain, such as qualified.trust.admin.eu for the trust list of qualified eIDAS signers. We do not want to have to download the entire trust list to check if somebody is a member, thus we assume every entry to have a particular search-key in form of a subdomain, for instance

PX2NO4LVPA4WHCBLYXHIKRWVRE.qualified.trust.admin.eu

We assume that we can lookup such a URL to get a trust list entry using a predicate

lookup(URL,TrustListEntry)

Note that this predicate triggers a server lookup, i.e., at the moment the policy checker (the TPL interpreter) reaches this lookup predicate, it interacts with the server. The predicate will succeed and return the respective TrustListEntry if this entry indeed exists, and otherwise the predicate will fail. It can thus act as a requirement in a policy.

Since a lookup query to a server is a time-consuming task (as compared to the other checks that are made locally), there is the risk that an inexperienced user specifies policies that get stuck in checking many queries repeatedly. For this reason, one could integrate a caching mechanism into the lookup predicate: all queries done over a certain period (say 5 minutes) are stored with their results, so that repeated queries within the time frame are answered from the cached result.

A certificate can now claim a trust list membership, for instance qualified signatures. For that, it shall include a field trustList that contains the URL for looking up the membership e.g. PX2NO4LVPA4WHCBLYXHIKRWVRE.qualified.trust.admin.eu.

Pitfall Note that there is a potential pitfall here: when the designer of a policy is not careful, it may happen that they just extract the trust list membership claim from a certificate and use the lookup function to just check the claim, for instance:

```
myfirstpolicy(Certificate) :−
  extract(Certificate,trustList,TrustList),
  lookup(TrustList,TrustListEntry),
  % followed by some checks on the TrustListEntry.
```

Even though the expectation of the modeler is that the trust list is, say, qualified.trust.admin.eu, there is nothing in this policy that checks that: this policy takes just whatever URL is contained in the document and queries that server, even if the URL is, say, trustme.attackerspace.tk. The checking of the entries on that trust list is probably irrelevant. While it may still be obvious in such a small policy, such an omission can easily slip into a more complex policy. An integrated development environment for TPL could to warn the user about such a specification.

For easily checking trust schemes, we thus assume another predicate trustscheme that relates a URL with the trust scheme it belongs to. For this, we assume a fixed association of trust schemes with particular URLs, so that users do not need to spell out URLs in their policies with all the potential vulnerabilities that come with that. For instance, we may have that

```
trustscheme(URL,eIDAS_qualified)
```

is true if and only if URL has "qualified.trust.admin.eu" as a suffix.

8 Further Examples

We now illustrate more advanced uses of TPL with the specification of seven scenarios inspired by the LIGHT^{est} architecture deliverable.

8.1 Boolean Trust Scheme without Translation

In the first scenario we consider an organization that receives documents and wants to check whether they are signed with eIDAS qualified signatures. Each document is therefore required to be signed and to come with a certificate proving that the signer is allowed to make eIDAS qualified signatures. A signer is allowed to make eIDAS qualified signatures if an issuer who is on the eIDAS trust list has given him this permission. Therefore, the certificate must be signed by such an issuer.

In this example the check made with the trust list is Boolean—either the issuer is on the trust list or she is not. Each entry of the trust list therefore only needs to contain a record of the public key of the issuer.

The organization can use the following TPL policy to make the checks:

```
checkQualifiedSignature(Document, Certificate, Signer) :−
  % Checking the certificate:
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, issuer, Issuer),
  extract(Certificate, bearer, Signer),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuerKey, PkIss),
  extract(Certificate, trustList, TrustMemClaim),
  % check the document was indeed signed with PkSig:
```

```
  verify_signature(Document, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss),
  % check the claimed trustlist membership is eIDAS qualified:
  trustscheme(TrustMemClaim, eIDAS_qualified),
  % check the Signer is really on the trustlist:
  lookup(TrustMemClaim, TrustListEntry),
  % check that the issuers key is indeed PkIss
  extract(TrustListEntry, pubKey, PkIss).
```

The verification of the signature with respect to a given public key we represent by the predicate verify_signature. The policy first requires that the format of the certificate is indeed a certificate for an eIDAS qualified signature and we extract the following information from it: the issuer (the entity who has signed the certificate), the bearer (the entity who owns the certificate), the pubKey (the public key of the bearer), the issuerKey (the public key of the issuer; this may be implicit), and finally the trustList which is a URL at which there should be an entry representing the claimed membership in the trust list.

Next, the policy verifies that the document is indeed signed with the the bearer's public key and the certificate with respect to the issuer's public key. We also check that the trust membership claim is really eIDAS qualified. Only after this, i.e. after all checks that can be done locally, we check the trust membership claim. This makes sense since the latter requires interaction with the corresponding server which we can spare us if any of the local checks fail. For the simple case of a Boolean trust list, TrustListEntry does not need to contain a lot of attributes, but it should contain at least the public key of the issuer, such that it can be verified to be the same as the issuer key that is recorded in the certificate.

The example shows that policies can be specified on an abstract level. In fact, on this level of concerns we do not specify the entire interaction with the DNS server and the checks that need to be performed on the response. Instead we only require, however the mechanism works in details, that the trust membership claim can be confirmed, i.e. that we can check that the issuer has the public key claimed in the certificate and is a member of the eIDAS qualified trust list.

8.2 Tuple-Based Trust Scheme without Translation

This example is a simple extension of the previous one, but this time the check with the trust list will be tuple-based, i.e. the trust list will assign an attribute to the trust lists entry, namely the attribute identityProofing. When this attribute is set to inPerson, it represents that the person has proved his identity by showing up in person.

It is easy to see that the policy is an extension of the previous policy since it extends it with a last line.

```
checkQualifiedSignatureInPerson(Document, Certificate, Signer) :-
  % Checking the certificate:
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, issuer, Issuer),
  extract(Certificate, bearer, Signer),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuerKey, PkIss),
  extract(Certificate, trustList, TrustMemClaim),
  % check the document was indeed signed with PkSig:
```

```
  verify_signature(Document, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss),
  % check the claimed trustlist membership is eIDAS qualified:
  trustscheme(TrustMemClaim, eIDAS_qualified),
  % check the Signer is really on the trustlist:
  lookup(TrustMemClaim, TrustListEntry),
  % check that the issuers key is indeed PkIss
  extract(TrustListEntry, pubKey, PkIss),
  % check that the signer showed
  extract(TrustListEntry, identityProofing, inPerson).
```

8.3 Trust Translation Scheme Scenario

As a next example we look at a Boolean trust translation scheme scenario. This is like the Boolean trust scheme scenario, but where the trust scheme is actually foreign (e.g. a Swiss trust scheme) and needs to be translated (e.g. into a European qualified signature). The beginning is similar again to the Boolean trust scheme; the difference is that we are using a variant of the trustscheme predicate, trustschemeX which we define afterwards.

```
checkQualifiedSignatureX(Document, Certificate, Signer) :-
  % Checking the certificate:
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, issuer, Issuer),
  extract(Certificate, bearer, Signer),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuerKey, PkIss),
  extract(Certificate, trustList, TrustMemClaim),
  % extract (potentially foreign) trust scheme:
  extract(Certificate, trustScheme, TrustScheme),
  % check the document was indeed signed with PkSig:
  verify_signature(Document, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss),
  % check the claimed trustlist membership is eIDAS qualified
  % or an equivalent one (hence the ...X):
  trustschemeX(TrustMemClaim, eIDAS_qualified),
  % check the Signer is really on the trustlist:
  lookup(TrustMemClaim, TrustListEntry),
  % Check that the issuers key is indeed PkIss
  extract(TrustListEntry, pubKey, PkIss).
```

The trustschemeX predicate checks that a trustlist membership claim is either directly to the trustlist we are looking for (here eIDAS_qualified), or belongs to a trustscheme that can be translated to eIDAS_qualified:

```
trustschemeX(Claim, DesiredScheme) :-
  % case 1: it is directly the desired scheme:
  trustscheme(Claim, DesiredScheme).
trustschemeX(Claim, DesiredScheme) :-
```

```
  % case 2: we can translate it to the desired scheme:
  encodeX(Claim, DesiredScheme, URL),
  lookup(URL, Entry),
  extract(Entry, translation, "equivalent").
```

Here the encodeX, given a claim for a foreign scheme and the name of the desired scheme, generates a URL for the trust translation scheme. For instance, suppose the Claim is a (hypothetical) Swiss scheme at URL "PX2NO4LV.admin.ch" and the DesiredScheme is eIDAS_qualified, then the URL shall be "admin__ch.Translation.signature.trust.eu" (i.e. escaping the base URL of the original scheme, and selecting the corresponding Translation scheme of eIDAS qualified). This URL then should refer to the entry for the Swiss scheme, if it exists, at eIDAS and we can check that the translation yields the result "equivalent". (This is a provision for more complex translation schemes that requires further attributes to fulfilled.)

Note that we could also make chains of translations. Imagine, for instance, that there is no direct translation from a Singaporean scheme to eIDAS, but the Singaporean scheme can be translated to the Swiss scheme which then again can be translated to eIDAS. We can formulate in TPL that such chains of translations should also be allowed. However, we cannot expect the verifier to find an appropriate set of translation hops automatically, so this would have to be provided as follows:

```
trustschemeChain(Claim, []).
trustschemeChain(Claim, [Hop|Hops]) :-
  encodeX(Claim, Hop, URL),
  lookup(URL, Entry),
  extract(Entry, translation, "equivalent"),
  trustschemeChain(URL, Hops).
```

Here we make use of the Prolog list notation where [] is the empty list and [X|Xs] represents a list that has at least one element X and a (possibly empty) rest list Xs.

8.4 Trust Scheme with Delegation Scenario

Let us now consider a simple delegation scenario: We have a Document that contains a purchase, signed by the proxy of a company. To this end, the company has issued a delegation Mandate containing at least the following information:

- A reference for the Mandator: In this example we assume this is a pointer to a trust membership claim of eIDAS.
- The public key (pkSig) of the proxy, such that the signature of the proxy can be verified.
- The purpose: In this example purchase.
- The authoritative delegation provider (DP).

The ATV is required to check that the delegation provider indeed has a valid entry containing a hash (HMandate) of the delegation mandate. This check ensures that the delegation has not been revoked by the mandator. The delegation provider is a server that has the authority of determining whether the mandate is valid or not. The delegation provider provides an entry containing the entire mandate in encrypted form plus a hash of the mandate. Therefore only the proxy of the mandate can read it, but the ATV that has received the mandate can check it with the hash.

The necessary checks to be performed can then be described by the following TPL specification:

```
checkQualifiedSignatureDelegation(Document, Mandate) :−
  % Check the Mandate fits the document:
  extract(Mandate, format, delegation),
  extract(Mandate, proxyKey, PkSig),
  verify_signature(Document, PkSig),
  extract(Mandate, purpose, purchase),
  % Check the mandator key (wrt Trust list):
  extract(Mandate, issuer, Mandator),
  trustscheme(Mandator, eIDAS_qualified),
  lookup(Mandator, TrustListEntry),
  extract(TrustListEntry, pubKey, PkIss),
  verify_signature(Mandate, PkIss),
  % Check the mandate is still valid:
  extract(Mandate, delegationProvider, DP),
  lookup(DP, DPEntry),
  extract(DPEntry, fingerprint, HMandate),
  verify_hash(Mandate, HMandate).
```

While the above example is rather simple, it could be generalized to more involved situations:

- In the example we assume the mandator is a pointed to a trust membership claim of eIDAS, but in general this could be more indirect, e.g. a certificate of the Mandator that was issued by an authority that is on the eIDAS trust list.
- In the example the mandate contains the public key of the proxy. Strictly speaking, one does not need the identity of the proxy here. Alternatively, the proxy could also prove its identity using an eIDAS trust scheme. Using a public key, however, gives pseudonymity for the proxy, i.e. while several purchases made with respect to the same mandate are of course linkable. In other words, the proxy can hide its identity behind a pseudonym, namely the public key, but there is no mechanism to hide that two purchases are made with respect to the same mandate.
- The purpose could be more fine-grained, e.g., allowing purchases only up to a certain limit.

8.5 University Case Study

Consider electronic admission to the PhD school of a university, say, the Technical University of Denmark. One of the requirements is that the candidates holds an MSc Degree (in a suitable subject, but we may leave this subject-aspect out for simplicity). The point of this scenario is that the applicant could prove this electronically, holding an electronically signed document from his or her alma mater, let us say university U.

First let us only consider the problem to check that the electronic diploma of the student is indeed from the claimed university. In Europe we can easily require that it is an eIDAS qualified signature, and like in the above scenarios we may allow for trust translation. Then we have an instance of the above document checking scenarios where the document is the student's diploma, and the certificate is the university's certificate.

So far however this only assures us that the issuer of the student's diploma is indeed the organization who is the owner of the university certificate. The problem is that any organization could self-apply the title "university". The idea is that one can build trust lists, for instance on a national government level, of which institutions are indeed recognized as universities, probably based on adhering certain academic standards. The European union can then choose to recognize

all such trust lists from its member states and from some non-EU countries based on bilateral agreements. This recognition can again be specified as a trust translation scheme:

```
realUniversity(Diploma, UniCertificate, Signer) :−
  % Checking the certificate:
  extract(UniCertificate, format, eIDAS_qualified_certificate),
  extract(UniCertificate, issuer, Issuer),
  extract(UniCertificate, bearer, Signer),
  extract(UniCertificate, pubKey, PkSig),
  extract(UniCertificate, issuerKey, PkIss),
  extract(UniCertificate, trustList, TrustMemClaim),
  % extract (potentially foreign) trust scheme:
  extract(UniCertificate, trustScheme, TrustScheme),
  % check the document was indeed signed with PkSig:
  verify_signature(Diploma, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(UniCertificate, PkIss),
  % check the claimed trustlist membership is on an EU recognized list
  % or an equivalent one (hence the ...X):
  trustschemeX(TrustMemClaim, eu_recognized_university),
  % check the Signer is really on the trustlist:
  lookup(TrustMemClaim, TrustListEntry),
  % Check that the issuers key is indeed PkIss
  extract(TrustListEntry, pubKey, PkIss).
```

Note that this policy does not check the contents of the Diploma, which may be in a local format and one has to still extract what kind of diploma it is.

9 Designing Translation Policies

LIGHT^{est} concerns executing trust policies, trust translation policies, and delegation policies. The task of LIGHT^{est} is *not* the design of such policies; this is often a result of political decisions (e.g. bilateral agreements between countries), related to issues that cannot yet be executed by a computer (e.g. a registration process where a person needs to be physically present), or the individual decision of a company (e.g. whether the company decides to trust entities of a particular trust scheme for orders up to a certain value). However, the language TPL gives us a possibility formalize policies and also—at least in parts—the relationships and concepts behind a credential (e.g. under which circumstances a certain level of assurance is satisfied in a given trust scheme). This formalization allows us to reasoning about policies and to use software tools that can perform such reasoning. In the example in the introduction, we already saw how we could use the ATV to do a bit of reasoning by asking the query "Who do we trust on delegation level 2?". By formalizing not only a policy but also the relationships and concepts behind it, we can ask similar questions on that level of abstraction. As we will see in this section, such formalizations can often be made without a lot of additional work— we simply need to formalize the relevant aspects in TPL. In this section, we expand on this "added value" of TPL.

We focus on the design of translation policies.

ISO LoA	eIDAS LoA
1	low
2	low
3	substantial
4	high

Table 1: A translation from levels of insurance in an ISO29115-scheme to levels of insurance in an eIDAS-scheme.

9.1 An Example

As a running example in this chapter, we use an eIDAS-scheme with three LoAs—Low, Substantial, and High—and an ISO29115-scheme with four—1, 2, 3, and 4. In this case, a translation policy can be represented as a table, stating what LoA in one scheme translates to what LoA in the other scheme.

As an example, consider a translation policy from an ISO29115-scheme to an eIDAS-scheme where

- a LoA of 1 or 2 translates to a Low LoA,
- a LoA of 3 translates to a Substantial LoA,
- and a LoA of 4 translates to a High LoA.

This translation policy is shown as represented by a table in table 1 and is straightforward to specify in TPL:

iso2eidas(iso_loa1, eidas_low).
iso2eidas(iso_loa2, eidas_low).
iso2eidas(iso_loa3, eidas_substantial).
iso2eidas(iso_loa4, eidas_high).

More complicated translations may be then expressed between schemes with several attributes. Keep in mind that this translation has a direction, namely from ISO to eIDAS, i.e., it is not supposed to be applied for translating from eIDAS to ISO since that does not even have a unique answer in every case. We can show that with TPL by making a query asking which ISO-levels are translated eIDAS level low:

iso2eidas(X, eidas_low).

This query yields the solutions X = iso_loa1 and X = iso_loa2. A translation in the direction eidas2iso could be defined in many ways, for example even in a way such that a Substantial LoA in the eIDAS-scheme is not enough to get a LoA of 3 in the ISO29115-scheme.

9.2 Designing Translations: the Rationale Behind a Policy

A translation only depends on information that is present in the credential—in the example from the previous subsection this was the level of assurance information in the given ISO credential. The reason why ISO level 2 is translated to eIDAS low and not eIDAS substantial is not formalized in this policy—and it should not be the concern of the ATV for instance. Rather, this is a concern of the design of the translation policy that may be based on all aspects of the two credential schemes, in particular the issuing process. This is because issuing depends on properties of the entity that the credential is being issued for, e.g., whether it is a person and whether this

person showed up in person to the issuing process. Such properties may have an influence on the assurance level that this person will obtain, but it may not be an attribute of the credential, so from the issued credential it is not (directly) visible whether it is a person who showed up in person. Thus, the policy cannot (directly) refer to such a property that is not reflected in the credential. It may however, be a design consideration for the translation policy. In a nutshell, the rationale could be:

> Credential C_A in scheme A should be translated to credential C_B in scheme B, if every entity who receives credential C_A in scheme A would get in scheme B the credential C_B or better.

This rationale requires several notions:

- There is a total ordering on the credentials of scheme B, expressing what is better. (The ordering should refer to only the ordinal aspects of credentials, not on other information like bearer name etc.)
- It requires that all properties of the issuing process of the credentials are sufficiently formalized and comparable between the two schemes. In the next sections, we show how to represent the above rationale in LIGHTest for any trust scheme that is sufficiently formalized in TPL.

9.3 The Issuing Process

Comparing credentials can be done by considering the details of issuance. Two schemes may have similar requirements for issuing credentials, and based on that decide to recognize the other scheme in a translation policy.

To talk about what requirements are met, we introduce the concept of a scenario as the input to the issuing process. A scenario assigns values to all properties relevant to the issuing process. A scheme can then decide to recognize a credential from a different scheme by evaluating all scenarios that could lead to that credential being issued.

Example 1. *In the following we see a scheme as a tuple $s_A = (A, issue_A)$, where A is the set of all credentials that s_A can issue, and $issue_A$ is a function, $S \to A$, that maps a scenario to a credential.*

As an example, to issue a credential to a person, a scheme may require the person to show up in person and present a passport. In this case, the scenario defines two Boolean properties, namely in_person and passport to indicate whether those requirements are met.

The function $issue_A$ of the scheme s_A is defined as follows.

$$issue_A(in\_person, passport) = \begin{cases} LoA\ High & \text{if } in\_person \text{ and } passport \\ LoA\ Low & \text{otherwise if } in\_person \\ \bot & \text{otherwise} \end{cases}$$

This scheme issues a certificate with a High LoA if both properties are true, and it issues a certificate with a Low LoA if only in_person is true. If none of these requirements are met, s_A issues no certificate, indicated with $\bot$ (undefined).

9.4 Translation

This section discusses how to specify a translation from a scheme s_A to another s_B. The translation is a function, $f_{AB} : A \to B$, from credentials of the first scheme to credentials of the second

scheme. The idea is to express f_{AB} in general, so that the same definition can be reused every time a new translation policy is to be defined.

One application opportunity is a tool that will automatically produce (or recommend) a translation policy. The only requirement would be to define the two input schemes in a compatible way by defining the issue-functions.

The idea is to define f_{AB} so that the credential it outputs is determined by the set of scenarios that the input-credential can be issued from. The problem is that these scenarios may correspond to more than one output-credential in the output-scheme.

More generally, if the credential a translates to the credential b, then it does not necessarily hold that the preimages of a and b under $issue_A$ and $issue_B$ are equal, i.e., that $\{x \mid issue_A(x) = a\} = \{x \mid issue_B(x) = b\}$. In other words, the set of entities x that qualify for credential a is not necessarily the same as the set of entities qualifying for b, simply because the issuing process may be based on slightly different criteria. Thus we cannot always ensure that the translation gives a truly equivalent credential; it is however recommended that the translation does *not* yield a credential that is *better* than the input to the translation. To that end, let us define a partial order on credentials that formulates what "better" actually means:

$$\begin{aligned} a >_{AB} b &\iff \{x \mid issue_A(x) = a\} \subsetneq \{x \mid issue_B(x) = b\} \\ a \geq_{AB} b &\iff \{x \mid issue_A(x) = a\} \subseteq \{x \mid issue_B(x) = b\} \end{aligned}$$

So $a > b$ says that a is strictly better than b, since the entities that satisfy a are a proper subset of those that satisfy b. The convention we recommend is thus:

Convention 1. *In general we recommend the following property of a translation f_{AB} from scheme A to scheme B:*

$$f_{AB}(a) = b \implies a \geq_{AB} b$$

According to this convention, a translation may downgrade a credential, but in general one wants to stay as close as possible to the original one:

Convention 2.

$$f_{AB}(a) = b \implies \forall b' >_{BB} b.\ a \not\geq_{AB} b'$$

This says that if a translates to b then there should not be any better level b' that would satisfy the convention: if that were the case then one should translate: $f_{AB}(a) = b'$.

In general, for a given a, there can be several values b and b' that satisfy the two conventions above, but then b and b' are incomparable. In purely ordinal schemes, however, there is always a uniquely defined maximum element.

Example 2. *As an example, consider the example of classes of driver's licenses: there may be a class for normal cars, one for motor cycles and one for trucks. The one for motor cycles entails the one for normal cars, and so does the one for trucks; however neither trucks subsume motor cycles nor vice-versa. Suppose now that we translate from another scheme for driver's licenses, where e.g. a military driver's license includes the ability to drive all kinds of vehicles, so both the translation to a motor cycle license and the translation to a truck license would be satisfying our conventions. Unless the target scheme has an equivalent for both motor cycle and truck, there is no best translation in general.*

Let us finally also consider an example that shows why in a negotiation, one may actually choose to not satisfy the conventions above:

Example 3. *Consider the space $S = \mathbb{B} \times \mathbb{B} \times \mathbb{N}$, representing the Boolean criteria in_person, passport and (for simplicity a natural number) the age.*

The schemes s_A and s_B are different in the condition for issuing the highest level credential:

$$issue_A(in\_person, passport, age) = \begin{cases} LoA\ 2 & \text{if } in\_person \text{ and } age \geq 20 \\ LoA\ 1 & \text{otherwise if } in\_person \text{ and } age \geq 18 \\ LoA\ 0 & \text{otherwise} \end{cases}$$

$$issue_B(in\_person, passport, age) = \begin{cases} LA & \text{if } in\_person \text{ and } age \geq 21 \\ LB & \text{otherwise if } in\_person \text{ and } age \geq 18 \\ \bot & \text{otherwise} \end{cases}$$

Following our conventions, there is no possible translation so that anyone gets an LA credential by translating from a credential issued under scheme s_A—even if the holder of the credential is actually 21 years old or older.

It may thus be a decision of a policy designer whether they want to break the convention here. Given that the age difference is only one year, one may decide to "turn a blind eye" to the small discrepancy and make a translation e.g. $f_{AB}(\text{LoA } 2) = \text{LA}$. However, one should be clear that this could have legal implications: suppose in the scheme B the legal drinking age is 21 and the LA credential is considered a reliable proof of that, then inserting this translation may break this trust relationship, e.g. so that LA credentials cannot be used after all for legally proving to be over 21.

10 Graphical TPL

We now finally present a new idea to graphically represent a large fragment of the TPL policies. Note that this language is aimed users who are experts in making policies, but do not necessarily computer science. Such users may want to work on the level of TPL, but need a shorthand at writing it.

We introduce here the *Graphical TPL*, or *GTPL* for short, using an example of an online platform for auction houses. We do not consider peer-to-peer auction houses like eBay that operate only online, but instead the platforms that have arisen to connect classical auction houses to the digital world without the need for every auction house to develop their own webpage and interface to the auctions. The auctions in questions may easily range up to thousands of Euros for a single item, which gives of course the classical problem of ensuring that the successful bidder indeed pays the sum they have bid. On the one hand, the auction house wants no entrance barrier for new customers who just "stumbled" upon an item by an Internet search; but on the other hand they want to avoid that, for instance, somebody practically anonymously bids on an item just to get the price up and then does not pay if they win the auction.

This is of course a classical *trust* problem. The classical solutions are that one has to bring references from other auction houses or a bank statement, or be present at the auction in person, proving one's identity before the auction starts. This example shows how to transfer these aspects to the digital world using LIGHT^{est} so that one can benefit from the large potential of the digital world without losing the security and trust guarantees of the classical auction houses.

10.1 Bidding Forms

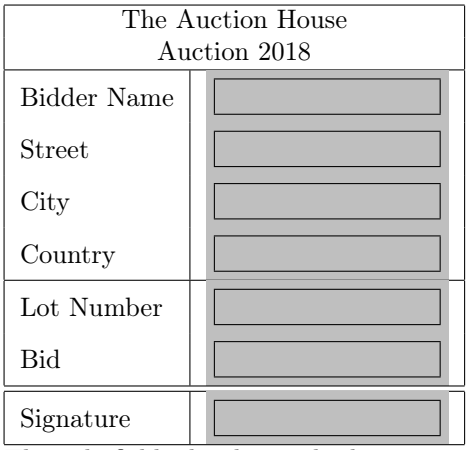
Classically, auction houses allow customers to bid via standard, non-digital, mail if they cannot be physically present as the auction. Bidders would for this purpose tell the auction house (per mail) a maximum bid for a particular item, and the auction house would accordingly act as if the bidder was present at the auction and bid up to the maximum bid. (After the auction, the successful bidders were notified and got an invoice, but only after payment, the auction house would send the items.)

For this purpose, each auction house would have their own bidding form. This is classically a simple paper form containing the personal information of the bidder and a list of items (the lot numbers and the maximum bid)—and of course a field where the bidder must sign the form. This would be sent to the auction house by mail.

Of course, the first step of digitalization was to have online catalogs, where one can click on items one wants to bid on, and enter a maximum amount. This would basically lead to an electronic version of the classical paper form and it would finally be transferred via https or simply email to the auction house. Such an electronic bidding form could look as follows (for simplicity we consider bidding only on a single item):

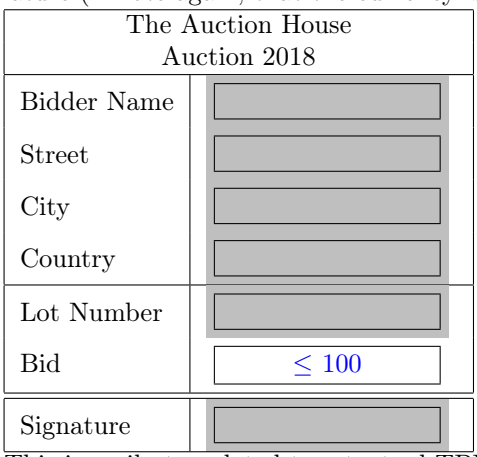
```
<AUCTIONHOUSEFORMAT
 auctionID="AUCTION18">
<bidder>...</bidder>
<address>
  <street>...</street>
  <city>...</city>
  <country>...</country>
</address>
<bid lotno="..."
     amount="...">
<signature>
jmj7l5rSw0yVbvlWAYkKYBwk
</signature>
</AUCTIONHOUSEFORMAT>
```

We have here already included a field for a digital signature, which is actually still optional in many of today's online bidding solutions. If used, it would be a digital signature on a hash of the document. The first step to integrate such a form in LIGHT^{est} is of course to define the abstract syntax, i.e., describe it as a list of attribute-value pairs and describe the relationship to the concrete syntax (parser and pretty-printer). This is the small amount of work to connect a new custom format into LIGHT^{est}. From there on, we can assume to have that format available. We also may assume, since formats are essentially lists of attribute-value pairs, that we can easily graphically represent them as a table with two columns as such:



The only fields that have a built-in meaning are the title fields (they identify the format name, and thus which fields are present) and the signature field: here we assume that this is really a digital signature algorithm. Additionally, to each field we may attach a type, e.g. that the Lot number and the Bid must be integer numbers, and we may have range restrictions on them. However, whether a bid is for instance in Euro or Swiss Francs is not a concern of the abstract syntax itself, but of the semantics of the form (although the syntax may have indications like "Bid (Euro)").

A First Policy Rule The basic idea is now that we can use this simple graphical representation of the "empty" form as a basis for describing policies. For instance, let us define as a first example the policy rule that the auction house wants to accept any bid up to 100 Euro, even without any signature (—note again, that the currency is implicit in the format):



This is easily translated to a textual TPL policy as follows:

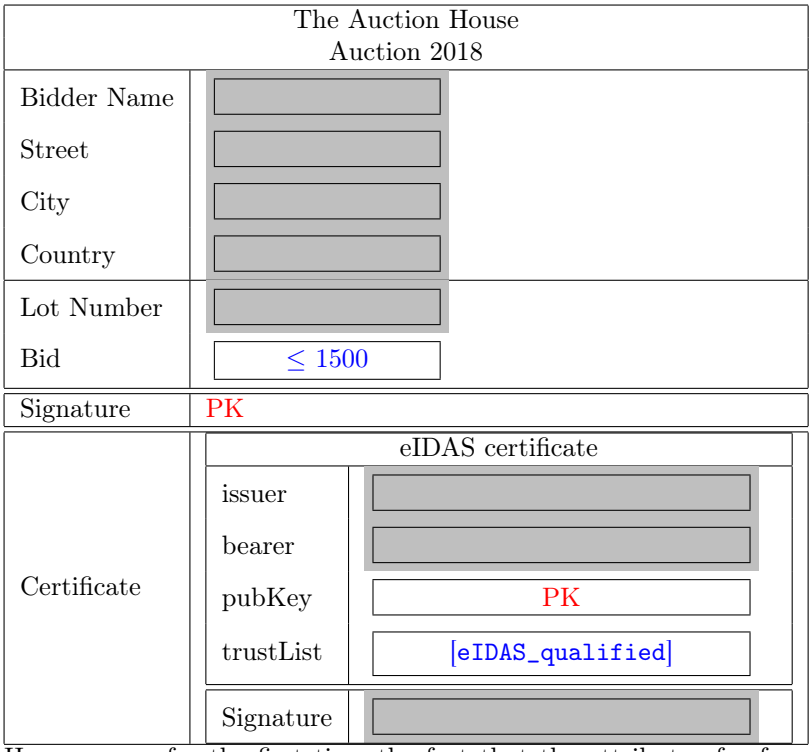
```
accept(FORM) :−
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID =< 100.
```

Here, a form is accepted if it is of the right format – this ensures that it has a bid-attribute—and that the BID is at most 100 Euro. Thus the basic idea is that policies can be formulated as a combination of constraints on the items of a form. Everything that is unconstrained remains an untouched (grey) field.

Checking Signatures As a next example, let us add that the auction house accepts any bid up to 1500 Euro, if it is signed by an eIDAS qualified signature. To that end we use that the Signature field has a distinguished meaning, namely that the ATV can check the signature with respect to a given public key and the document. Of course, we do not wish to specify here any concrete public key, but rather that it is a key that belongs to a particular trust scheme, here eIDAS qualified. We thus introduce here the notation [eIDAS qualified], i.e., the name of a trust scheme in square brackets to indicate that.

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	$\leq$ 1500
Signature	[eIDAS qualified]

The notation [eIDAS qualified] here is a short notation for a verification process that actually has quite a number of details. In particular, this implies that the signature comes with an eIDAS certificate (or has a pointer to where one can be obtained). Then the signature has to be verified against the public key of the certificate. The certificate itself is signed, and that signature has also to be checked, namely by checking that the issuer indeed is on the trust list and with the public key that verifies the certificate. Let us describe also this relationship in GTPL:



Here, we use for the first time the fact that the attribute of a form may itself be a form, namely the certificate. Note that we have here depicted the Certificate field as an additional field of the Auction house form; this may of course be organized in a different way, e.g. as a link to a URL where the information can be obtained. Note that we have here put the Certificate field below the signature which is a convention of GTPL to indicate that the signature does not span the Certificate.

With all the details clarified, we can translate this policy into TPL as follows:

```
accept(FORM) :−
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID =< 1500,
  extract(FORM, signature, Signature),
  extract(FORM, certificate, Certificate),
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, pubKey, PK),
  verify_signature(Signature, PK),
  extract(Certificate, issuerKey, PkIss),
  verify_signature(CertSignature, PkIss),
  extract(Certificate, trustList, TrustMemClaim),
  extract(Certificate, signature, CertSignature),
  trustscheme(TrustMemClaim, eIDAS_qualified),
  lookup(TrustMemClaim, TrustListEntry),
  % check that PkIss is indeed in the TrustListEntry!
  extract(TrustListEntry, pubKey, PkIss).
```

Allowing Trust Translation As a next step, we want to additionally accept also signatures outside eIDAS if they are deemed equivalent via a translation scheme of eIDAS. To that end, we introduce the notion of $\approx$ for a trust scheme, meaning equivalence modulo a translation (where we rely on the trust translation schemes provided by the authority of the target scheme):

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	$\in$ `EU_EFTA_list`
Lot Number	
Bid	$\leq$ 1500
Signature	$\approx$[eIDAS qualified]

We have here introduced yet another notation: since value of the country attribute must come from a finite list of values, we may define subsets (like `EU_EFTA_list`) and check membership. So this policy would accept bids up to 1500 Euro from EU/EFTA-countries, as long they have an eIDAS qualified certificate, or equivalent.

```
accept(FORM) :−
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID =< 1500,
  extract(FORM, signature, Signature),
  extract(Signature, format, sigform),
  extract(Signature, certificate, Certificate),
  % The following is the standard check of the certificate
  % and its trust membership claim specialized for eIDAS
  extract(Certificate, format, eIDAS_qualified_certificate),
  extract(Certificate, issuer, Issuer),
  extract(Certificate, bearer, Signer),
  extract(Certificate, pubKey, PkSig),
  extract(Certificate, issuerKey, PkIss),
  extract(Certificate, trustList, TrustMemClaim),
  % extract (potentially foreign) trust scheme:
  extract(Certificate, trustScheme, TrustScheme),
  % check the document was indeed signed with PkSig:
  verify_signature(FORM, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, PkIss),
  % check the claimed trustlist membership is eIDAS qualified or equivalent:
  trustschemeX(TrustMemClaim, eIDAS_qualified),
  % check the Signer is really on the trustlist:
  lookup(TrustMemClaim, TrustListEntry),
  % check that the issuers key is indeed PkIss
  extract(TrustListEntry, pubKey, PkIss).
```

Similarly, we can have the rule that we accept any bid up to 1000 EUR, as long as the signature is eIDAS qualified or can be translated to eIDAS. Here we do not put restrictions on the country:

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	$\leq$ 1000
Signature	≈[eIDAS qualified]

Note that all the policies we formulate are put together by *disjunction*, i.e., a bid is accepted if any of the rules matches. The order of the rules only determines in which order they are checked, so it makes sense to put the most common cases first, and the more specific cases later. Also, we here check the easy-to-check constraints first, e.g. constraints on the amount of the bid, while the more "expensive" checks that involve a server-lookup are put last.

Custom-built Trust Schemes LIGHT^{est} does not only support a fixed number of pre-existing schemes like eIDAS, but also allows for defining new trust schemes. As an example, there may be a platform for auction houses that acts as a unifying service to auction houses, i.e., auction houses can register there and thus share a customer base where customers do not have to register themselves or have certificates. Suppose that this auction house platform, let us call it `platform.de`, has its own bidding form to allow users to place bids. This one has now has the particular auction house as a field, say auctionID of the form:

User Name	
User Level	
Street	
City	
Country	
AuctionID	`Auctionhouse18`
Lot Number	
Bid	
Signature	`platform.de`

Note that here we fill in a concrete value for the Signature field, namely the domain name `platform.de` as a means to say that the signature must be with a key from `platform.de`. It is thus not the users signing the bid, but the auction house platform signing on behalf of the user. This is actually the status how it is done today in most such platforms. Given that users could

build up a reputation and given that the auction house trusts the platform, the auction house may formulate their own trust policy with respect to such bids arriving from the platform, e.g. let us specify that we accept bids up to 5000 Euro if the user has level premium:

User Name	
User Level	PREMIUM
Street	
City	
Country	
AuctionID	Auctionhouse18
Lot Number	
Bid	$\leq$ 5000
Signature	platform.de

The TPL translation would simply be:

```
accept(FORM) :−
  extract(FORM, format, thePlatformFormat),
  extract(FORM, userlevel, premium),
  extract(FORM, auctionID, auctionhouse18),
  extract(FORM, bid, BID), BID =< 5000,
  extract(FORM, signature, Signature),
  extract(Signature, format, sigform),
  verify(FORM, pk_platform_de).
```

Here, for simplicity, we have assumed that the auction house already has the corresponding public key of the platform (pk_platform_de), but this may again be obtained via certificates based, e.g., on eIDAS.

Now to make this more flexible in the future, and to avoid that users have to reveal all their bids to the platform, the platform can issue certificates for their users and thus establish a trust list:

The Auction House Auction 2018	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	$\leq$ 5000
Signature	PK

Platform.de User Certificate	
User Name	
User Level	PREMIUM
Street	
City	
Country	
Publickey	PK
Signature	platform.de

Here we have the bidding sheet of the auction house and a certificate from the platform vouching for the user. The link between them is the variable PK, the public key of the user:

the bidding sheet has to verify with respect to this public key, and the certificate has to be established with respect to this.

The translation into TPL is as follows:

```
accept(FORM) :−
  extract(FORM, format, theAuctionHouse2018format),
  extract(FORM, bid, BID), BID =< 5000,
  extract(FORM, signature, Signature),
  extract(Signature, format, sigform),
  extract(Signature, certificate, Certificate),
  % The following is the standard check of the certificate
  % and its trust membership claim specialized for eIDAS
  extract(Certificate, format, thePlatformCertificate),
  extract(Certificate, userlevel, premium),
  extract(Certificate, issuer, Issuer),
  extract(Certificate, bearer, Signer),
  extract(Certificate, pubKey, PkSig),
  % check the document was indeed signed with PkSig:
  verify_signature(FORM, PkSig),
  % check the certificate is indeed signed with PkIss:
  verify_signature(Certificate, pk_platform_de).
```

Again, the public key here is fixed to pk_platform_de, as we are talking about a fixed issuer.

Trust Scheme for References As a final point in this example, we want to show how a full-fledged tuple-based trust scheme can also be used. While so far the auction house could directly rely on the platform as a trusted party, we want to consider a scenario where it is more indirect. In the classical auction landscape, a bidder who is known at one auction house may get a reference for use at another auction house where he or she is bidding for the first time. Classically, that makes only sense if the auction house that gives the reference is actually known and can be verified. This is again where trust schemes can help, since the auction house platform may establish a trust list of auction houses that are valid for giving letters of reference, so our auction house may choose to rely on them without even knowing them. The following policy expresses exactly that: the auction house accepts bids up to 5000 Euro when there is a certificate of reference by an auction house on the trust list `refs.platform.de`:

Auction House'18	
Bidder Name	
Street	
City	
Country	
Lot Number	
Bid	≤ 5000
References	User Reference
	Issued by
	User Name
	Street
	City
	Country
	Publickey: PK
	Rating: PREMIUM
	Signature: [refs.platform.de]
Signature	PK

Note that we use here a URL to refer to a trust scheme; such a technical aspect may be otherwise "hidden" by a constant e.g. references_platform_trustlist.

The translation to TPL would be as follows:

```
accept(FORM) :-
  extract(FORM,format,theAuctionHouse2018format),
  extract(FORM,bid,BID), BID =< 5000,
  extract(FORM,signature,PkSig),
  % check the document was indeed signed with PkSig:
  verify_signature(FORM, PkSig),
  extract(Reference,format,platform_user_reference),
  extract(Reference,publickey,PkSig),
  extract(Reference,rating,premium),
  extract(Reference,signature,Signature),
  extract(Reference,issuerKey,PkIss),
  extract(Reference,trustList,TrustMemClaim),
  trustscheme(TrustMemClaim,refs_platform_de),
  verify_signature(Reference, PkIss),
  lookup(TrustMemClaim,TrustListEntry),
  extract(TrustListEntry, pubKey, PkIss).
```

GTPL Translation Semantics We now give the semantics of GTPL by a translation function from GTPL to standard TPL (which of course already has a semantics). The GTPL input, although graphical, is actually also represented here in a textual form as follows: every form is represented as an expression F(attributes) where F is the name of the form (e.g. eIDAS_certificate) and attributes is a list of (attribute-value) pairs, where the first component of each pair is an attribute name like signature and the second component is one of the allowed expressions of GTPL, namely either BLANK (the gray fields), a concrete value, a variable, a comparison operator along with a concrete value or variable, or another form in the same syntax. In the special case of the trustlist attribute, we also allow the attribute to be [value] or ≈[value] where value is the name of a trust scheme (like eIDAS). The semantics starts with calling $[\![F(Attlist)]\!]^{Toplevel}$ for a form, and it will generate a TPL clause accept(Form) :− ... . that holds true if Form satisfies all the requirements specified in $F(Attlist)$. The full specification is given in Figure 1.

The GTPL translation procedure starts from the top level form, e.g., the online form of the auction house in the previous example. The first semantic function starts a new rule of the trust policy that first checks the format of the form and then the function continues by translating the list of attributes and their corresponding values contained in the form.

If any attribute is unconstrained, i.e., an untouched BLANK field, then we continue with the rest of the attributes in the list until the attribute list becomes empty. If for an attribute a particular value or a variable is specified, then the translation is to use the extract predicate for this attribute and have the value or variable as a third argument, i.e., if the third argument is a value the translated policy will check that the attribute in the format has that value, and if the third argument is the first occurrence of a variable then that variable is bound to the value corresponding to the attribute in the format. If for an attribute an operator and term was specified, then the translation is to extract the attribute into a new variable and compare the variable to the term.

The attribute signature is a special case: in this case we verify that the form is indeed signed with the private key of the corresponding public key X. In fact, we assume here that the form itself provides in case of a signature enough information to determine which public key corresponds to the signing key, the part of the text that is signed, and the signature itself. If, for instance, the public key is actually not part of the information conveyed in this form, then there is a complication that we omitted in the semantics here: in this case there must be another place (outside given the form) related to the policy where the public key is obtained. In such cases one must delay the checking of the signature until reaching the step where this public key is extracted.

In case that the attribute value is itself a form (i.e. a subform), we extract the subform and continue with the subform's attribute list with the same translation procedure recursively.

Finally, trust_list is also a special attribute: the value of such an attribute shall use the notation [trustscheme] to indicate the requirement that the certificate issuer must be on the given trustscheme, or shall use the notation ≈[trustscheme] to indicate the more relaxed requirement that the signature is either on this trustscheme or on a foreign trust scheme that can be translated to trustscheme. In the first case, the attribute of trust_list is extracted, a URL, and we use the predicate trustcheme to verify that the URL indeed belongs to the desired trust scheme; then we look up the URL to get a trust list entry, to extract the public key from and verify the certificate with. The second case with trust translation is almost identical, except for using the predicate trustchemeX instead to allow for translations.

$[\![F(Attlist)]\!]^{Toplevel} =$
accept(Form) :−
$[\![F(Attlist)]\!]^{Formlevel}_{Form}$.
where Form is a new variable.

$[\![F(Attlist)]\!]^{Formlevel}_{Form} =$
extract(Form,format,F)
$[\![F(Attlist)]\!]^{Attlevel}_{Form}$

$[\![(attname, BLANK) : Attlist]\!]^{Attlevel}_{Form} = [\![Attlist]\!]^{Attlevel}_{Form}$

$[\![(signature, X) : Attlist]\!]^{Attlevel}_{Form} =$
,verify_signature(Form,X)
$[\![Attlist]\!]^{Attlevel}_{Form}$

$[\![(attname, t) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,attname,t)
$[\![Attlist]\!]^{Attlevel}_{Form}$
where t is either a term or a value.

$[\![(attname, Comp\ t) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,attname,Valuevar),Valuevar Comp t
$[\![Attlist]\!]^{Attlevel}_{Form}$
where Valuevar is a new variable and t is either a term or a value.

$[\![(attname, R(Attlist_s)) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,attname,Subform),
$[\![R(Attlist_s)]\!]^{Formlevel}_{Subform}, [\![Attlist]\!]^{Attlevel}_{Form}$
where Subform is a new variable.
$[\![(trust\_list, [Value]) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,trust_list,TrustMemClaim)
,trustscheme(TrustMemClaim,Value)
,lookup(TrustMemClaim,TrustListEntry)
,extract(TrustListEntry,pubKey,PK)
,verify_signature(Form,PK)
$[\![Attlist]\!]^{Attlevel}_{Form}$
where TrustMemClaim, TrustListEntry, and PK are new variables.

$[\![(trust\_list, \approx [Value]) : Attlist]\!]^{Attlevel}_{Form} =$
,extract(Form,trust_list,TrustMemClaim)
,trustschemeX(TrustMemClaim,Value)
,lookup(TrustMemClaim,TrustListEntry)
,extract(TrustListEntry,pubKey,PK)
,verify_signature(Form,PK)
$[\![Attlist]\!]^{Attlevel}_{Form}$
where TrustMemClaim, TrustListEntry, and PK are new variables.

Figure 1: The translation function from GTPL to TPL.

11 Conclusions

We have introduced the trust policy language (TPL) for describing trust policies, trust schemes, trust translation schemes, and trust delegation schemes. The main aim was to define a language that is conceptually simple and clear, while at the same time powerful enough for our purposes. TPL is powerful in the sense that with its executable semantics it serves as a programming language in which we can write policies as programs. TPL is simple in the sense that its language is simply that of horn clauses which from a logical perspective has a simple semantics namely that of first-order logic. A particular benefit of using Prolog-style Horn clauses is the direct executability of TPL policies. This also makes it feasible to implement the machinery of LIGHT^{est} on this basis and to prove implementations correct. Another benefit is accountability: one can easily prove that a decision was made correctly adhering to a given policy.

For simple business cases with rather straightforward policies one may argue that not much is gained from the powerfulness and simplicity of TPL. On the other hand, for complicated policies the powerfulness of the language may be necessary to even state them, and the simple semantics may be needed to resolve disagreements over the meaning of a policy. Notice also that powerfulness and simplicity come for free—the powerfulness and simplicity do not make it difficult to express simple policies as shown by the numerous examples in this technical report. Furthermore, there is work on usability in LIGHT^{est}—an example being GTPL. This language is designed with a wide variety of people in mind, who do not necessarily have a background in computer science. That it is based on the more powerful TPL comes at no extra cost, but ensures that it has a simple logical basis.

References

[1] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorization Language, 2006. Microsoft Research Technical Report MSR–TR–2006–120.

[2] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard: Reference Manual*. Springer, 2012.

[3] Y. Gurevich and I. Neeman. DKAL: Distributed-Knowledge Authorization Language. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 149–162, 2008.

[4] B. K. Mejborn. ASN.2: A model-driven approach to secure protocol implementation, 2016. Bachelor Thesis DTU, 2016, available upon request.

[5] S. Mödersheim and G. Katsoris. A sound abstraction of the parsing problem. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 259–273, 2014.